26 | Join Hints指南：不同场景下，如何选择Join策略?

2021-05-12 吴磊

Spark性能调优实战 进入课程 >

讲述：吴磊
时长 19:24 大小 17.78M

你好，我是吴磊。

在数据分析领域，数据关联可以说是最常见的计算场景了。因为使用的频率很高，所以 Spark 为我们准备了非常丰富的关联形式，包括 Inner Join、Left Join、Right Join、Anti Join、Semi Join 等等。

搞懂不同关联形式的区别与作用，可以让我们快速地实现业务逻辑。不过，这只是基础，要想提高数据关联场景下 Spark 应用的执行性能，更为关键的是我们要能够深入理解 Join 的实现原理。

所以今天这一讲，我们先来说说，单机环境中 Join 都有哪几种实现方式，它们的优劣势分别是什么。理解了这些实现方式，我们再结合它们一起探讨，分布式计算环境中 Spark 都

支持哪些 Join 策略。对于不同的 Join 策略，Spark 是怎么做取舍的。

Join 的实现方式详解

到目前为止，数据关联总共有 3 种 Join 实现方式。按照出现的时间顺序，分别是嵌套循环连接（NLJ，Nested Loop Join ）、排序归并连接（SMJ，Shuffle Sort Merge Join）和哈希连接（HJ，Hash Join）。接下来，我们就借助一个数据关联的场景，来分别说一说这 3 种 Join 实现方式的工作原理。

假设，现在有事实表 orders 和维度表 users。其中，users 表存储用户属性信息，orders 记录着用户的每一笔交易。两张表的 Schema 如下：

复制代码

```
// 订单表orders关键字段
userId, Int
itemId, Int
price, Float
quantity, Int

// 用户表users关键字段
id, Int
name, String
type, String //枚举值，分为头部用户和长尾用户
```

我们的任务是要基于这两张表做内关联（Inner Join），同时把用户名、单价、交易额等字段投影出来。具体的 SQL 查询语句如下表：

复制代码

```
//SQL查询语句
select orders.quantity, orders.price, orders.userId, users.id, users.name
from orders inner join users on orders.userId = users.id
```

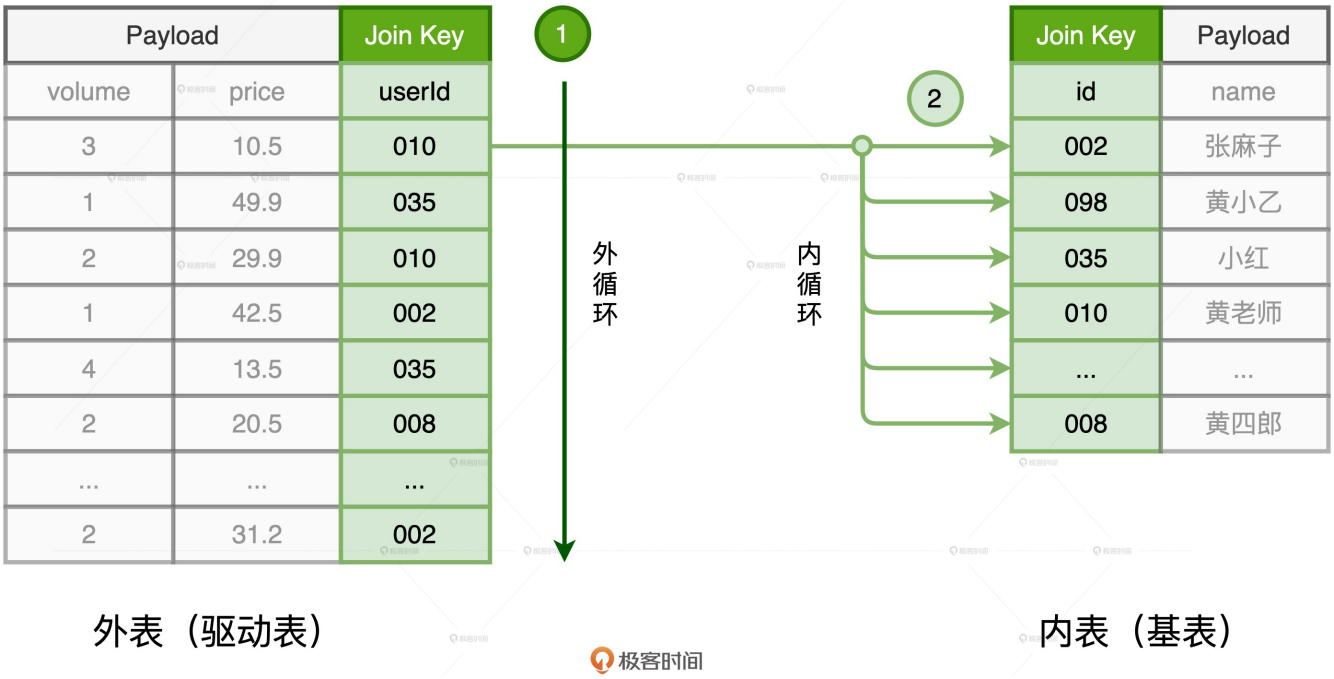
那么，对于这样一个关联查询，在 3 种不同的 Join 实现方式下，它是如何完成计算的呢？

NLJ 的工作原理

对于参与关联的两张数据表，我们通常会根据它们扮演的角色来做区分。其中，体量较大、主动扫描数据的表，我们把它称作外表或是驱动表；体量较小、被动参与数据扫描的

表，我们管它叫做内表或是基表。那么，NLJ 是如何关联这两张数据表的呢？

NLJ 是采用“嵌套循环”的方式来实现关联的。也就是说，NLJ 会使用内、外两个嵌套的 for 循环依次扫描外表和内表中的数据记录，判断关联条件是否满足，比如例子中的 orders.userId = users.id，如果满足就把两边的记录拼接在一起，然后对外输出。

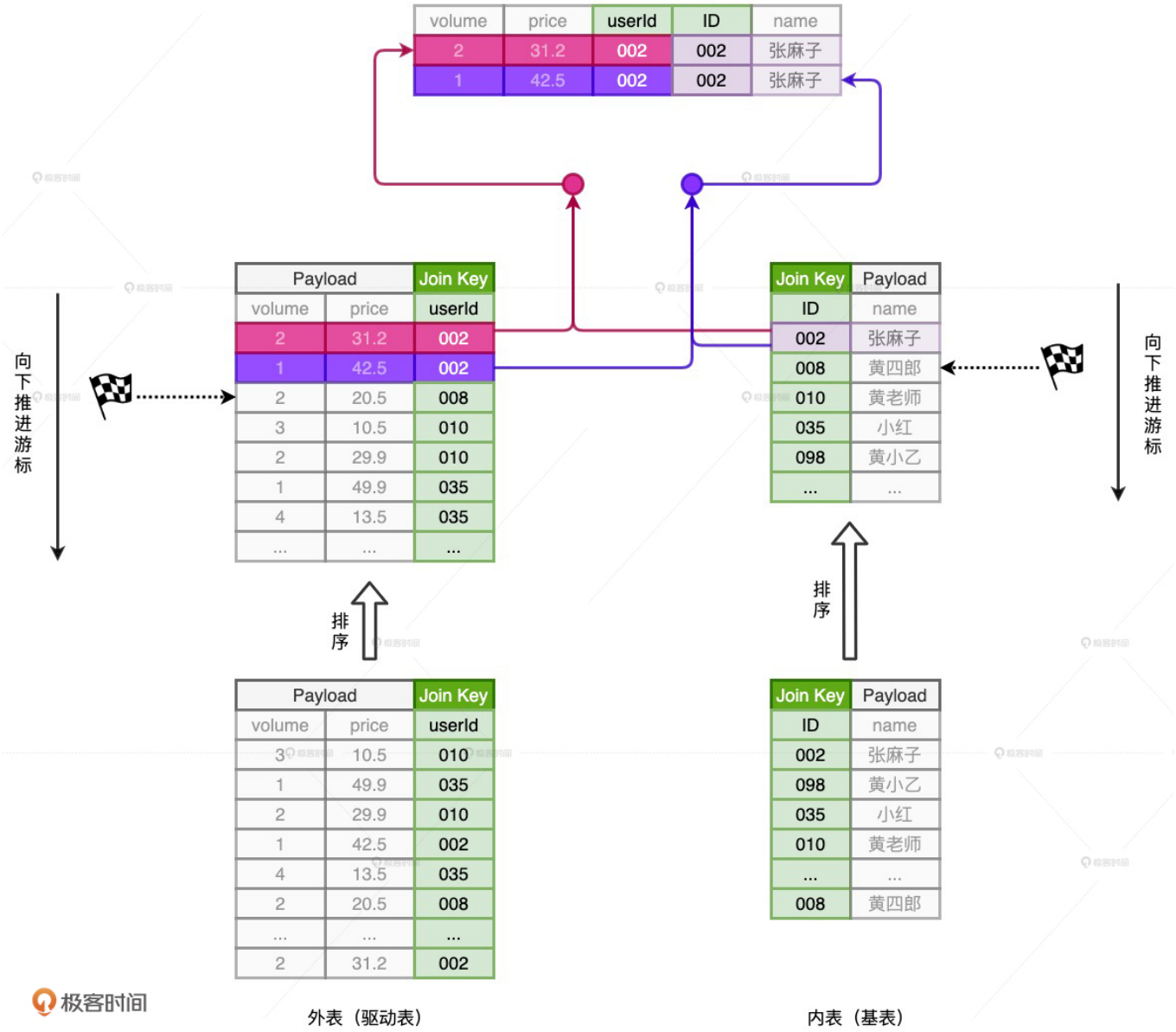


Nested Loop Join示意图

在这个过程中，外层的 for 循环负责遍历外表中的每一条数据，如图中的步骤 1 所示。而对于外表中的每一条数据记录，内层的 for 循环会逐条扫描内表的所有记录，依次判断记录的 Join Key 是否满足关联条件，如步骤 2 所示。假设，外表有 M 行数据，内表有 N 行数据，那么 **NLJ 算法的计算复杂度是 O(M * N)**。不得不说，尽管 NLJ 实现方式简单而又直接，但它的执行效率实在让人不敢恭维。

SMJ 的工作原理

正是因为 NLJ 极低的执行效率，所以在它推出之后没多久之后，就有人用排序、归并的算法代替 NLJ 实现了数据关联，这种算法就是 SMJ。**SMJ 的思路是先排序、再归并。**具体来说，就是参与 Join 的两张表先分别按照 Join Key 做升序排序。然后，SMJ 会使用两个独立的游标对排好序的两张表完成归并关联。



Sort Merge Join示意图

SMJ 刚开始工作的时候，内外表的游标都会先锚定在两张表的第一条记录上，然后再对比游标所在记录的 Join Key。对比结果以及后续操作主要分为 3 种情况：

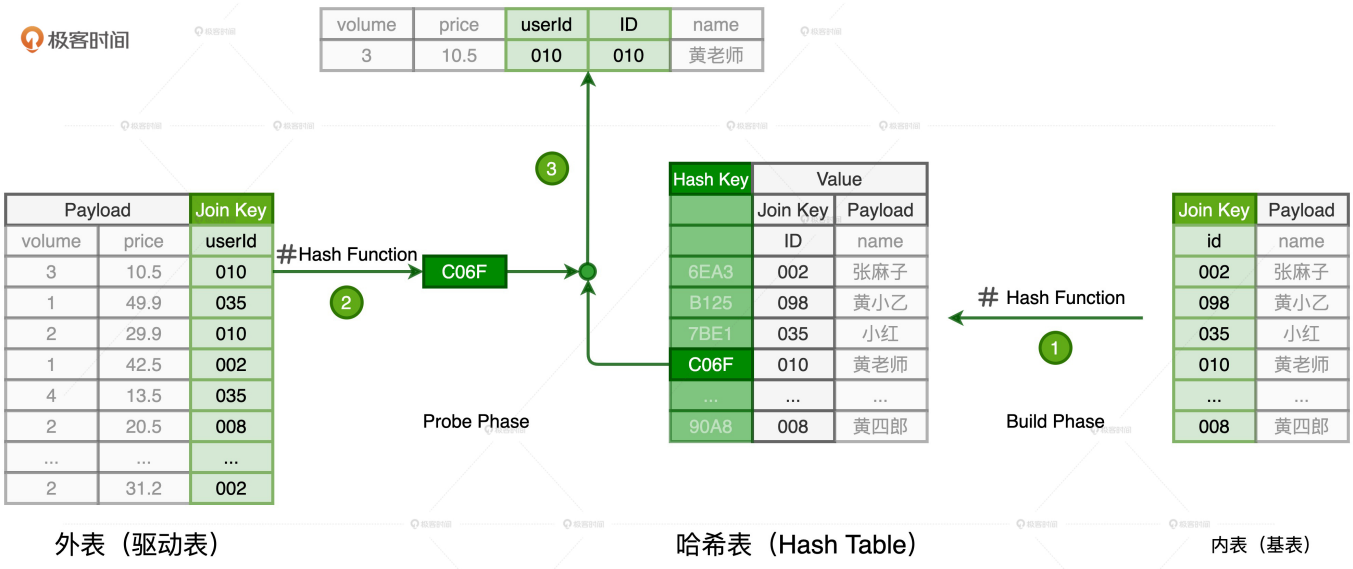
1. 外表 Join Key 等于内表 Join Key，满足关联条件，把两边的数据记录拼接并输出，然后把外表的游标滑动到下一条记录
2. 外表 Join Key 小于内表 Join Key，不满足关联条件，把外表的游标滑动到下一条记录
3. 外表 Join Key 大于内表 Join Key，不满足关联条件，把内表的游标滑动到下一条记录

SMJ 正是基于这 3 种情况，不停地向下滑动游标，直到某张表的游标滑到头，即宣告关联结束。对于 SMJ 中外表的每一条记录，由于内表按 Join Key 升序排序，且扫描的起始位置为游标所在位置，因此 **SMJ 算法的计算复杂度为 O(M + N)**。

不过，SMJ 计算复杂度的降低，仰仗的是两张表已经事先排好序。要知道，排序本身就是一项非常耗时的操作，更何况，为了完成归并关联，参与 Join 的两张表都需要排序。因此，SMJ 的计算过程我们可以用“先苦后甜”来形容。苦的是要先花费时间给两张表做排序，甜的是有序表的归并关联能够享受到线性的计算复杂度。

HJ 的工作原理

考虑到 SMJ 对排序的要求比较苛刻，所以后来又有人提出了效率更高的关联算法：HJ。HJ 的设计初衷非常明确：**把内表扫描的计算复杂度降低至 O(1)**。把一个数据集合的访问效率提升至 O(1)，也只有 Hash Map 能做到了。也正因为 Join 的关联过程引入了 Hash 计算，所以它叫 HJ。



HJ 的计算分为两个阶段，分别是 Build 阶段和 Probe 阶段。在 Build 阶段，基于内表，算法使用既定的哈希函数构建哈希表，如上图的步骤 1 所示。哈希表中的 Key 是 Join Key 应用（Apply）哈希函数之后的哈希值，表中的 Value 同时包含了原始的 Join Key 和 Payload。

在 Probe 阶段，算法遍历每一条数据记录，先是使用同样的哈希函数，以动态的方式（On The Fly）计算 Join Key 的哈希值。然后，用计算得到的哈希值去查询刚刚在 Build 阶段创建好的哈希表。如果查询失败，说明该条记录与维度表中的数据不存在关联关系；如果查询成功，则继续对比两边的 Join Key。如果 Join Key 一致，就把两边的记录进行拼接并输出，从而完成数据关联。

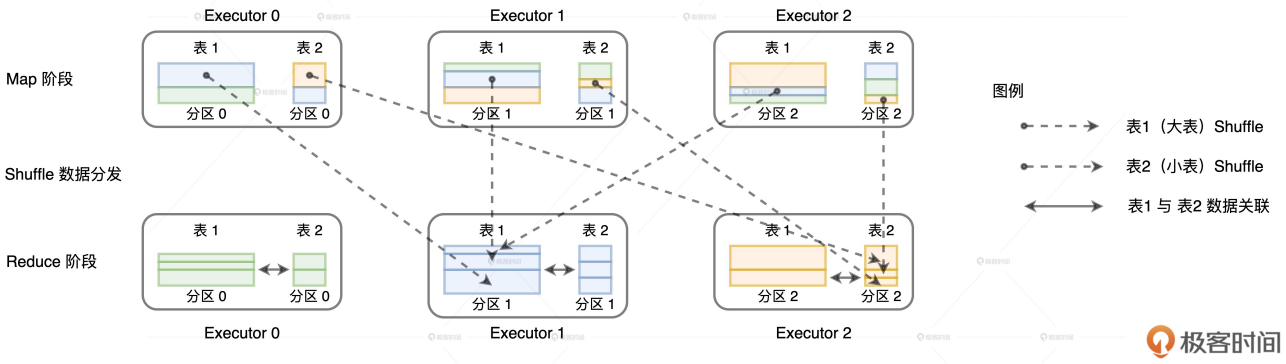
分布式环境下的 Join

掌握了这 3 种最主要的数据关联实现方式的工作原理之后，在单机环境中，无论是面对常见的 Inner Join、Left Join、Right Join，还是不常露面的 Anti Join、Semi Join，你都能对数据关联的性能调优做到游刃有余了。

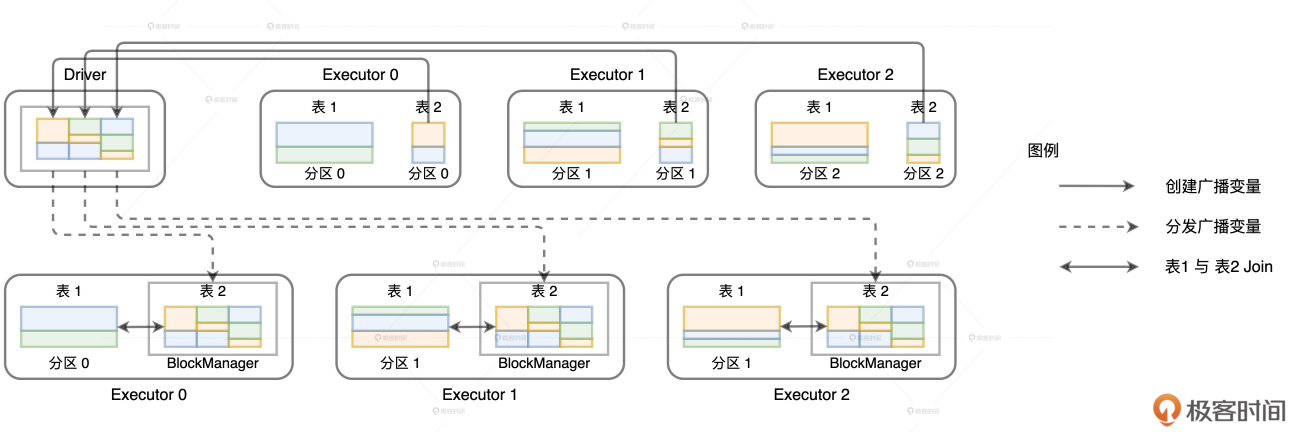
不过，你也可能会说：“Spark 毕竟是个分布式系统，光学单机实现有什么用呀？”

所谓万变不离其宗，实际上，**相比单机环境，分布式环境中的数据关联在计算环节依然遵循着 NLJ、SMJ 和 HJ 这 3 种实现方式，只不过是增加了网络分发这一变数**。在 Spark 的分布式计算环境中，数据在网络中的分发主要有两种方式，分别是 Shuffle 和广播。那么，不同的网络分发方式，对于数据关联的计算又都有哪些影响呢？

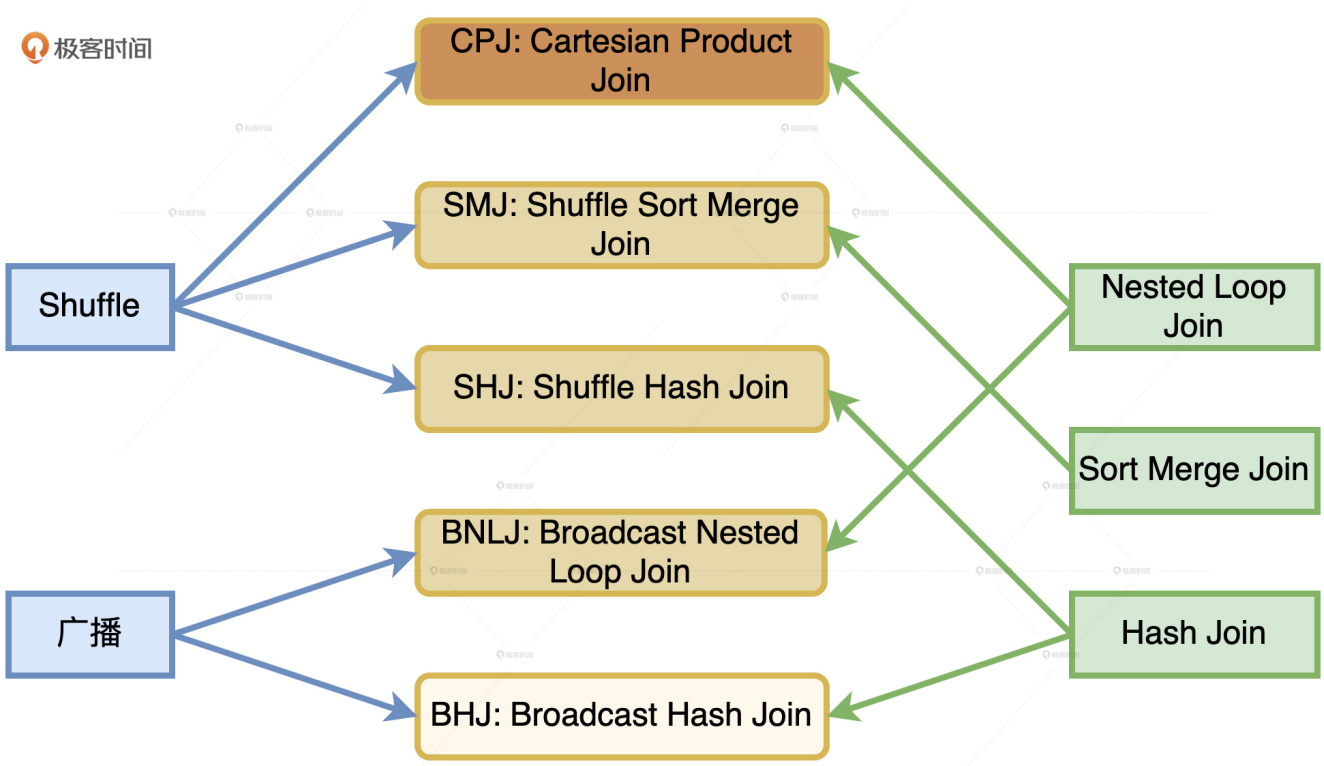
如果采用 Shuffle 的分发方式来完成数据关联，那么外表和内表都需要按照 Join Key 在集群中做全量的数据分发。因为只有这样，两个数据表中 Join Key 相同的数据记录才能分配到同一个 Executor 进程，从而完成关联计算，如下图所示。



如果采用广播机制的话，情况会大有不同。在这种情况下，Spark 只需要把内表（基表）封装到广播变量，然后在全网进行分发。由于广播变量中包含了内表的**全量数据**，因此体量较大的外表只要“待在原地、保持不动”，就能轻松地完成关联计算，如下图所示。



不难发现，结合 Shuffle、广播这两种网络分发方式和 NLJ、SMJ、HJ 这 3 种计算方式，对于分布式环境下的数据关联，我们就能组合出 6 种 Join 策略，如下图所示。



这 6 种 Join 策略，对应图中 6 个青色圆角矩形，从上到下颜色依次变浅，它们分别是 Cartesian Product Join、Shuffle Sort Merge Join 和 Shuffle Hash Join。也就是采用 Shuffle 机制实现的 NLJ、SMJ 和 HJ，以及 Broadcast Nested Loop Join、Broadcast Sort Merge Join 和 Broadcast Hash Join。

从执行性能来说，6 种策略从上到下由弱变强。相比之下，CPJ 的执行效率是所有实现方式当中最差的，网络开销、计算开销都很大，因而在图中的颜色也是最深的。BHJ 是最好的分布式数据关联机制，网络开销和计算开销都是最小的，因而颜色也最浅。此外，你可能也注意到了，Broadcast Sort Merge Join 被标记成了灰色，这是因为 Spark 并没有选择支持 Broadcast + Sort Merge Join 这种组合方式。

那么问题来了，明明是 6 种组合策略，为什么 Spark 偏偏没有支持这一种呢？要回答这个问题，我们就要回过头来对比 SMJ 与 HJ 实现方式的差异与优劣势。

相比 SMJ，HJ 并不要求参与 Join 的两张表有序，也不需要维护两个游标来判断当前的记录位置，只要基表在 Build 阶段构建的哈希表可以放进内存，HJ 算法就可以在 Probe 阶段遍历外表，依次与哈希表进行关联。

当数据能以广播的形式在网络中进行分发时，说明被分发的数据，也就是基表的数据足够小，完全可以放到内存中去。这个时候，相比 NLJ、SMJ，HJ 的执行效率是最高的。因此，在可以采用 HJ 的情况下，Spark 自然就没有必要再去用 SMJ 这种前置开销比较大的方式去完成数据关联。

Spark 如何选择 Join 策略?

那么，在不同的数据关联场景中，对于这 5 种 Join 策略来说，也就是 CPJ、BNLJ、SHJ、SMJ 以及 BHJ，Spark 会基于什么逻辑取舍呢？我们来分两种情况进行讨论，分别是等值 Join，和不等值 Join。

等值 Join 下，Spark 如何选择 Join 策略?

等值 Join 是指两张表的 Join Key 是通过等值条件连接在一起的。在日常的开发中，这种 Join 形式是最常见的，如 t1 inner join t2 on **t1.id = t2.id**。

在等值数据关联中，Spark 会尝试按照 BHJ > SMJ > SHJ 的顺序依次选择 Join 策略。在这三种策略中，执行效率最高的是 BHJ，其次是 SHJ，再次是 SMJ。其中，SMJ 和 SHJ 策略支持所有连接类型，如全连接、Anti Join 等等。BHJ 尽管效率最高，但是有两个前提条件：一是连接类型不能是全连接（Full Outer Join）；二是基表要足够小，可以放到广播变量里面去。

那为什么 SHJ 比 SMJ 执行效率高，排名却不如 SMJ 靠前呢？这是个非常好的问题。我们先来说结论，相比 SHJ，Spark 优先选择 SMJ 的原因在于，SMJ 的实现方式更加稳定，更不容易 OOM。

回顾 HJ 的实现机制，在 Build 阶段，算法根据内表创建哈希表。在 Probe 阶段，为了让外表能够成功“探测”（Probe）到每一个 Hash Key，哈希表要全部放进内存才行。坦白说，这个前提还是蛮苛刻的，仅这一点要求就足以让 Spark 对其望而却步。要知道，在不同的计算场景中，数据分布的多样性很难保证内表一定能全部放进内存。

而且在 Spark 中，SHJ 策略要想被选中必须要满足两个先决条件，这两个条件都是对数据尺寸的要求。**首先，外表大小至少是内表的 3 倍。其次，内表数据分片的平均大小要小于广播变量阈值。**第一个条件的动机很好理解，只有当内外表的尺寸悬殊到一定程度时，HJ

的优势才会比 SMJ 更显著。第二个限制的目的是，确保内表的每一个数据分片都能全部放进内存。

和 SHJ 相比，SMJ 没有这么多的附加条件，无论是单表排序，还是两表做归并关联，都可以借助磁盘来完成。内存中放不下的数据，可以临时溢出到磁盘。单表排序的过程，我们可以参考 Shuffle Map 阶段生成中间文件的过程。在做归并关联的时候，算法可以把磁盘中的有序数据用合理的粒度，依次加载进内存完成计算。这个粒度可大可小，大到以数据分片为单位，小到逐条扫描。

正是考虑到这些因素，相比 SHJ，Spark SQL 会优先选择 SMJ。事实上，在配置项 spark.sql.join.preferSortMergeJoin 默认为 True 的情况下，Spark SQL 会用 SMJ 策略来兜底，确保作业执行的稳定性，压根就不会打算去尝试 SHJ。开发者如果想通过配置项来调整 Join 策略，需要把这个参数改为 False，这样 Spark SQL 才有可能去尝试 SHJ。

不等值 Join 下，Spark 如何选择 Join 策略?

接下来，我们再来说说不等值 Join，它指的是两张表的 Join Key 是通过不等值条件连接在一起的。不等值 Join 其实我们在以前的例子中也见过，比如像查询语句 t1 inner join t2 on **t1.date > t2.beginDate and t1.date <= t2.endDate**，其中的关联关系是依靠不等式连接在一起的。

由于不等值 Join 只能使用 NLJ 来实现，因此 Spark SQL 可选的 Join 策略只剩下 BNLJ 和 CPJ。在同一种计算模式下，相比 Shuffle，广播的网络开销更小。显然，在两种策略的选择上，Spark SQL 一定会按照 BNLJ > CPJ 的顺序进行尝试。当然，BNLJ 生效的前提自然是内表小到可以放进广播变量。如果这个条件不成立，那么 Spark SQL 只好委曲求全，使用笨重的 CPJ 策略去完成关联计算。

开发者能做些什么?

最后，我们再来聊聊，面对上述的 5 种 Join 策略，开发者还能做些什么呢？通过上面的分析，我们不难发现，Spark SQL 对于这些策略的取舍也基于一些既定的规则。所谓计划赶不上变化，预置的规则自然很难覆盖多样且变化无常的计算场景。因此，当我们掌握了不同 Join 策略的工作原理，结合我们对于业务和数据的深刻理解，完全可以自行决定应该选择哪种 Join 策略。

Join策略	策略缩写	Join Hints	适用场景
Broadcast Hash Join	BHJ	broadcast broadcastjoin	内表较小，通常在2GB左右为宜，最大不能超过8GB
Shuffle Merge Sort Join	SMJ	merge mergejoin	等值Join下，Spark SQL的兜底策略，不需要开发者特别设置
Shuffle Hash Join	SHJ	shuffle_hash	当大表超过小表3倍以上，且小表数据分布比较均匀，SHJ比SMJ效率更佳
Broadcast Nested Loop Join	BNLJ	无	没有提供Join Hints，开发者无能为力，只能靠Spark SQL自行判断
Cartesian Product Join	CPJ	shuffle_replicate_nl	不等值Join下，Spark SQL的兜底策略，不需要开发者特别设置

极客时间

在最新发布的 3.0 版本中，Spark 为开发者提供了多样化的 Join Hints，允许你把专家经验凌驾于 Spark SQL 的选择逻辑之上。**在满足前提条件的情况下，如等值条件、连接类型、表大小等等，Spark 会优先尊重开发者的意愿，去选取开发者通过 Join Hints 指定的 Join 策略。**关于 Spark 3.0 支持的 Join Hints 关键字，以及对应的适用场景，我把它们总结到了如上的表格中，你可以直接拿来参考。

简单来说，你可以使用两种方式来指定 Join Hints，一种是通过 SQL 结构化查询语句，另一种是使用 DataFrame 的 DSL 语言，都很方便。至于更全面的讲解，你可以去第 13 讲看看，这里我就不多说了。

小结

这一讲，我们从数据关联的实现原理，到 Spark SQL 不同 Join 策略的适用场景，掌握这些关键知识点，对于数据关联场景中的性能调优至关重要。

首先，你需要掌握 3 种 Join 实现机制的工作原理。为了方便你对比，我把它们总结在了下面的表格里。

Join实现方法	工作原理
Nested Loop Join	在外表、内表之上，使用嵌套的两层For循环来实现数据关联，效率最低，算法复杂度为O(M * N)
Sort Merge Join	首先，要求参与关联的数据表有序。其次，对于两个有序表，算法使用游标滑动的方式来完成外表和内表的关联计算，算法复杂度为O(M + N)
Hash Join	分为两个阶段。在Build阶段，算法使用哈希算法，在内表之上建立哈希表。在Probe阶段，算法遍历外表的每一条记录，使用同样的哈希算法计算Join Key的哈希值，通过查找哈希表的方式完成关联计算。由于哈希表的查询效率是O(1)，因此Hash Join算法的计算复杂度是O(M)

极客时间

掌握了 3 种关联机制的实现原理，你就能更好地理解 Spark SQL 的 Join 策略。结合数据的网络分发方式（Shuffle 和广播），Spark SQL 支持 5 种 Join 策略，按照执行效率排序就是 BHJ > SHJ > SMJ > BNLJ > CPJ。同样，为了方便对比，你也可以直接看下面的表格。

关联关系	Join策略的选择顺序
等值Join	Spark SQL优先选择BHJ，其次是SMJ，再次是SHJ。其中，SHJ执行效率更高，但是内表数据全部放入内存的要求过于苛刻，SMJ可以充分利用磁盘来完成排序与归并关联，因此，Spark SQL默认选择SMJ作为兜底策略，完成等值Join的关联计算
不等值Join	因为不等值Join只能依赖NLJ算法，因此内表满足广播阈值条件的情况下，在这种关联场景中，Spark SQL会优先选择BNLJ。否则，Spark SQL只能使用笨重的CPJ来兜底去完成关联计算

极客时间

最后，当你掌握了不同 Join 策略的工作原理，结合对于业务和数据的深刻理解，实际上你可以自行决定应该选择哪种 Join 策略，不必完全依赖 Spark SQL 的判断。

Spark 为开发者提供了多样化的 Join Hints，允许你把专家经验凌驾于 Spark SQL 的选择逻辑之上。比如，当你确信外表比内表大得多，而且内表数据分布均匀，使用 SHJ 远比默认的 SMJ 效率高得多的时候，你就可以通过指定 Join Hints 来强制 Spark SQL 按照你的意愿去选择 Join 策略。

每日一练

1. 如果关联的场景是事实表 Join 事实表，你觉得我们今天讲的 Sort Merge Join 实现方式还适用吗？如果让你来设计算法的实现步骤，你会怎么做？
2. 你觉得，不等值 Join 可以强行用 Sort Merge Join 和 Hash Join 两种机制来实现吗？为什么？

期待在留言区看到你的思考和答案，我们下一讲见！

提建议

学习推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | Spark 3.0（二）：DPP特性该怎么用？

下一篇 27 | 大表Join小表：广播变量容不下小表怎么办？

精选留言 (9)

写留言

kingcall

2021-05-12

回答：
问题1：个人觉得SMJ 就是用在两张大表上的关联才有意思啊，也就是事实表 Join 事实表，但是这里要求是等值关联，如果是不等值关联就只能CPJ
问题2：可以分情况讨论一下，但是肯定是可以实现的
1. != 这样的关联，Sort Merge Join 和 Hash Join 都是不划算的，但是是可以实现...

展开

作者回复: 到位~

1. Shuffle Sort Merge Join，对于“大表Join大表”来说，是最稳定的实现方式，不过关于“大表Join大表”，咱们还是有一些调优手段的，这块可以关注28、29讲哈~

2. 确实，排序和构建hash table本身成了开销，在不等值的情况下，已经没有任何意义。

2

斯盖丸

2021-05-13

老师一般SQL的教程里都是join的优化方法之一是小表join大表，但我看Spark里你都是大表join小表，在大数据里谁join谁要紧吗，感觉好像无所谓？

展开

作者回复: 没有区别哈~

小表Join大表，掉过来不就是大表Join小表么？其实不论谁Join谁，其实优化器都是会做区分的。不过通常来说，在形式上，我们把外表（驱动表）放左边，而内表（基表）放右边，所以专栏里面，在描述的时候，都是“大表Join小表”。

1 1

巧克力黑

2021-05-22

老师你好，产生了几个疑问，请帮忙看看我理解的对么
1、从MapReduce阶段来看JOIN，带Shuffle的Join应该都发生在Reduce阶段？

2、经过Shuffle后的数据不就已经是排序的么，这样子使用SMJ是不是比SHJ少了Hash计算，也减少构建Hash的内存开销。...

展开

辰
2021-05-19

老师，斯盖丸同学的大表join小表，在spark中，大表作为驱动表放在左边，那和放在右边有效率影响吗？还是不成文的规范，第二点，在hive中，小表是放在左边的，如果足够小的话，hive会自动把小表放进内存中，相当于广播变量了

展开

作者回复: 大小表放哪边其实没关系的，Spark SQL在优化阶段会自行判断。实际上两张表重要的是“大、小”，而不是“左、右”，对于Spark SQL优化引擎来说，大小重要，左右不重要。我们之所以区分左右，主要是为了叙述方便而已~

斯盖丸
2021-05-17

老师，问个生产上的问题，我把autoBroadcastThreshold值调到了3g，然后就遇到了spark.driver.maxResultSize超出最大值的错误。该值默认1g，我调到了5g，已经大于autoBroadcastThreshold的3g了，可为什么还报错，请问生产上这两个参数的最佳实践该怎么互相指定呢?

展开

作者回复: 这个听上去有点蹊跷，也就是说，maxResultSize=5G，autoBroadcastThreshold=3G，在这种情况下，在尝试创建广播变量的时候报错，对吧？报的错误是maxResultSize超出阈值限制?

结合现象的话，我觉得有一种可能是，对于数据集大小，Spark判断其小于3G，但实际上它的尺寸是6G，这个时候Spark还是会尝试创建广播变量，但是实际广播变量尺寸很大，超过了maxResultSize，所以会报你说的错误。

所以这里的关键是，你要广播的数据集，大小究竟是多大？这个可以用我们之前说的方法去计算、验证一下，然后再来判断，咱们的猜测是不是对的。对于数据集尺寸的预估，Spark确实有“看走眼”的时候，并不是每次都能做到十分精确。

aof
2021-05-16

第二题，如果使用SMJ实现不等值Join，比如在大于或小于的关联条件下，排序还是有些

用处的，因为拿一个表的Join key去另一个表扫描的时候，遍历到不满足条件的记录时，就可以不用继续遍历了。但是如果不排序，每次都是无脑全部遍历，虽然在计算时间复杂度的时候可能还是一样的，但是，按概率来讲，多数时候，实际的遍历记录是不一样的，还是有性能提升的。

展开

作者回复: 我同意你的说法，就是从统计概率来说，在扫描的时候会有一定的提升，不过SMJ有个前提哦，那就是参与Join的两表要提前排序，这个开销还是蛮大的~

Fendora范东_

2021-05-12

问题回答
1.等值条件时，内外表任意大小都可以用SMJ，只不过当内表比较小或者内表平均分片小于阈值时，有性能更好的BHJ与SHJ可以选择。特别是大表join大表时，不满足使用其他两个的条件，SMJ就是最优解。
...

展开

作者回复: 分析的很到位~

1. Shuffle sort merge join 在Shuffle Joins里面确实是最稳定的实现方式了。不过对于“大表Join大表”来说，其实咱们还是有一些优化手段的，这块可以关注下28、29讲哈~

2. 没错，就是这么回事~

zxk

2021-05-12

1. 可以对 Sort Merge Join 做一个变种，例如一个表排序，一个表不排序，不排序的表作为驱动表，排序的表可以通过二分查找等方式快速定位驱动表的 Join Key。
2. 也可以强行实现，但 Sort Merge Join 方面的排序就会变得毫无意义，同时 Sort Merge Join、Hash Join 的时间复杂度也并未降低，反而带来了额外的排序开销与内存开销。

展开

作者回复: 满分💯

第一题答得尤其好，这道题的初衷就是鼓励大家发散思维，大开脑洞，去思考更多的实现方式~

你说的实现方式，我觉得蛮有意思，算法复杂度介于NLJ和SMJ之间，也就是O（M*logN），前置开销是一次排序，这种实现其实还是能适配一些场景的~ 666~

第二题分析的很到位，强行用SMJ的时候，排序就成了“戴斗笠打雨伞”，多此一举。（其实我特别想说，脱了裤子放屁，不过不太文雅，哈哈）

Bennan
2021-05-12

1.两张事实表最好的等值连接方式就是smj，可以让map端输出的时候先进行排序，reduce拉取数据的时候就可以对两个表的多个数据流进行join操作
2可以强行使用smj和hj，但是这样并没有意义，因为最后join还是m*n的复杂度(当时如果是大于或者小于的连接方式，在进行连接的可以优化一下)，而smj会带来额外的排序开销，hj要求内存能够放得下并且需要构建hash表

展开

作者回复: 没错，大表Join大表最稳定的方式，确实就是Shuffle Sort Merge Join。同时如果Map阶段计算按照（partitionId，join key）排好序的话，后续Reduce阶段可以直接做SMJ的计算，相当于前置条件已经满足。

第二题说的没错，排序、构建哈希表本身成了开销、多此一举~